Y.M. Teo and X.B. Wang, *ALiCE: A Scalable Runtime Infrastructure for High Performance Grid Computing*, Proceedings of IFIP International Conference on Network and Parallel Computing, pp. xx, Springer-Verlag Lecture Notes in Computer Science, Wuhan, China, October 2004.

ALiCE: A Scalable Runtime Infrastructure for High Performance Grid Computing

Y. M. Teo[1,2] and X.B. Wang[1,2]

[1]*Department of Computer Science, National University of Singapore, Singapore 117543*
[2]*Singapore-MIT Alliance, 4 Engineering Drive 3, Singapore 117576*
email: teoym@comp.nus.edu.sg

Abstract

This paper discusses the design and implementation of ALiCE, a Java-based grid computing middleware to facilitate the development and deployment of generic grid applications on heterogeneous shared computing resources. The ALiCE layered grid architecture comprises of a core layer that provides the basic services for control and communication within a grid. Programming template in the extensions layer provides a distributed shared-memory programming abstraction that frees the grid application developer from the intricacies of the core layer and the underlying grid system. This template can be used to develop specialized applications and high-level programming models for specific problem domains. Performance of a distributed Data Encryption Standard (DES) key search problem on two grid configurations are discussed.

1. Introduction

Grid computing [7, 13] is an emerging technology that enables the utilization of shared resources distributed across multiple administrative domains, thereby providing dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities [9] in a collaborative environment. These resources can include supercomputers, storage systems, data sources and special classes of devices. Clustering and using them as a single unified resource forms a networked virtual supercomputer [10] is popularly known as a *computational grid* [9]. Grids can be used to provide computational, data, application, information services, and consequently, knowledge services, to the end users [2], which can either be a human or a process. The selection and aggregation of a specific resource to a grid depends on factors such as its availability, capability, performance, cost of use and users' quality-of-service (QoS) requirements [17].

A grid systems developer generally faces five main challenges in the course of engineering a reliable grid infrastructure:

- Firstly, grid environments are characteristically dynamic, and the grid must adapt to the aggregation and dislodging of resources any time in operation.
- Secondly, most applications are inherently sequential and difficult to parallelize effectively, so there is a need to provide middleware and feasible higher-level programming models to facilitate application development in a distributed environment.
- Thirdly, since resources are being shared across the grid, reliable code safety and security mechanisms must be incorporated to protect them from malicious codes and users respectively.
- Fourthly, data communications between components are very prevalent in a grid, and it is necessary to introduce efficient and flexible techniques for the transfer of large datasets on the grid.
- Lastly, the grid may be composed of networks with vastly different latencies, bandwidths and configurations, and resources have different capabilities, which may render it a daunting task to implement efficient resource scheduling and allocation algorithms [6] that provide acceptable performance and with minimal overhead.

Considerable efforts by many international research collaborations have seen the developments of various projects associated with grid computing. These projects can be hierarchically categorized as *integrated grid systems*, *application(s)-driven* efforts and *middleware* [2]. NetSolve [5] is one example of an integrated grid system. It is a client/server application designed to solve computational science problems in a wide-area distributed environment. A NetSolve client communicates, using Matlab or the Web, with the server, which can adopt any scientific package in the computational kernel. This system is analogous to FAFNER [26], a massively parallel factoring application that marked the crucial milestone amongst the pioneer projects in grid computing. The European DataGrid [20] is a highly distinguished instance of an application-driven grid effort. Its objective is to develop a grid dedicated to the analysis of large volumes of data obtained from scientific experiments, and to establish productive collaborations between scientific groups based in different geographical locations. Middlewares developed for grid computing include Globus [10, 12], GridSim [4], Legion [18] and JXTA [25]. The Globus metacomputing toolkit attempts to facilitate the construction of computational grids by providing a *metacomputing abstract machine*: a set of loosely coupled basic services that can be used to implement higher-level components and applications. These services include a communications module based on the Nexus communication library [14], Global Toolkit Resource Allocation Manager (GRAM) [11] for resource management, Metacomputing Directory Service (MDS) [8] for access to instantaneous information about the grid and GridFTP [1] for efficient transfer of files. Globus is realigning its toolkit with the emerging OGSA grid standard [12]. GridSim, on the other hand, is a toolkit for modeling and simulation of grid resources and application scheduling, and offers a complete solution for the simulation of different categories of heterogeneous resources, users, applications, resource brokers and schedulers. Legion is a metacomputing toolkit that treats all hardware and software components in the grid as objects that are able to communicate with each other through method invocations. Like Globus, Legion pledges to provide users with the vision of a single virtual machine. JXTA, on the other hand, implements a library of components to facilitate P2P computing in a distributed environment.

This paper presents ALiCE (*Adaptive scaLable Internet-based Computing Engine*), a grid computing *core middleware* designed for secure, reliable and efficient execution of distributed applications on any Java-compatible platform. Our main design goal is to provide developers of grid applications with a user-friendly programming environment that does away with the hassle of implementing the grid infrastructure, thus enabling them to concentrate solely on their application problems. The middleware encapsulates services for compute and data grids, resource scheduling and allocation, and facilitates application development with a straightforward programming template.

The remainder of this paper is structured as follows. Section 2 describes the design of ALiCE including its architecture, runtime system and object communication architecture. Section 3 presents ALiCE implementation. Section 4 discusses the ALiCE template-based distributed shared-memory programming model. Section 5 evaluates the performance of ALiCE using a key search problem. Our concluding remarks are in Section 6.

2. System Design

2.1. The Objective of ALiCE

Several projects, such as Globus [10] and Legion [24], attempt to provide users with the vision of a single abstract machine for computing by the provision of core/user-level middleware encapsulating fundamental services for inter-entity communications, task scheduling and management of resources. Likewise, ALiCE is a portable middleware designed for developing and deploying general-purpose grid applications and application programming models. However, unlike Globus toolkit which is a collection of grid tools, ALiCE is a grid system.

ALiCE is designed to meet a number of design goals. ALiCE achieves *flexibility* and *scalability* through its capability to support the execution of multiple applications concurrently and the presence of multiple clients within the grid. *Modularity* is attained by the clean segregation of the grid engine into core and extension services, analogous to the basic services for security, resource management and data transfer provided in Globus [10]. ALiCE enables grid applications deployment on all operating systems and hardware platforms due to its implementation in the *platform independent* Java language, unlike systems such as Condor [15], which is C-based and executes only on WinNT and Unix platforms. ALiCE also offers an API to achieve *generic* runtime *infrastructure support*, allowing the deployment of any distributed application: this is a major feature a middleware has to provide, which distinguishes itself from application-driven efforts that are problem-specific, like SETI@Home [27]. Besides the deployment of Java applications, we are developing execution support for applications written in other commonly used development languages, such as C and C++, so as to achieve *generic developmental language support*. *Anonymity* and *security* are well handled in ALiCE, as only authenticated machines are allowed to operate in an ALiCE grid, and the information pertaining to each participating machine in the grid is undisclosed to all other machines.

2.2. Architecture

The AliCE grid architecture as shown in Figure 1 comprises of three constituent layers, *ALiCE Core*, *ALiCE Extensions* and *ALiCE Applications and Toolkits*, built upon a set of Java technologies and operating on a grid fabric.

The *grid fabric* encompasses the physical hardware components and networks within the grid. It includes all participating computational resources that run ALiCE processes. The fabric may comprise of homogeneous workstation PCs tightly coupled in a cluster, heterogeneous machines loosely connected across a LAN or WAN, or a hybrid of both configurations.

The ALiCE system is written in Java and implemented using *Java technologies* including Sun Microsystems' Jini^{TM} and JavaSpaces^{TM} [21] for resource discovery services and object communications within a grid. It also works with GigaSpaces^{TM} [16], an industrial implementation of JavaSpaces. To support the execution of applications regardless of their developmental language, ALiCE uses Java Native Interface (JNI) to enable the runtime infrastructure to invoke non-Java code.

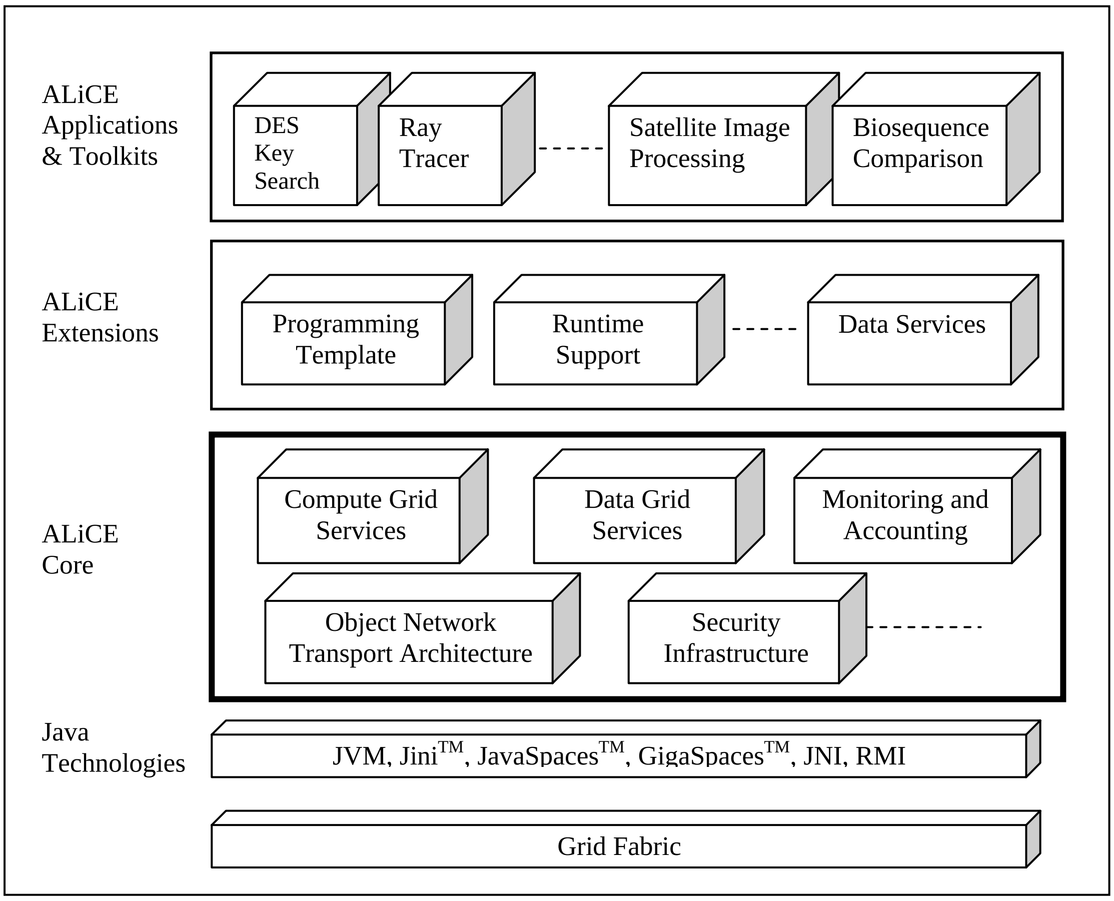

Figure 1: ALiCE Layered Grid Architecture

The *ALiCE core* layer encompasses the basic services used to develop grids. Compute Grid Services (CGS) include algorithms for resource management, discovery and allocation, as well as the scheduling of compute tasks. Data Grid Services (DGS) are responsible for the management of data accessed during computation, locating the target data within the grid and ensuring multiple copy updates where applicable. The security service is concerned with maintaining the confidentiality of information within each node and detecting malicious code. The major security features include user authentication and data encryption, and a code safety technique adopted is application *sandboxing* [29]. Object communication is performed via our Object Network Communication Architecture (ONTA) that coordinates the transfer of information-encapsulated objects within the grid. ONTA will be discussed in greater detail in Section 2.4. Besides these grid foundation services, a monitoring and accounting service is also included.

The *ALiCE extensions* layer encompasses the ALiCE runtime support infrastructure for application execution and provides the user with a distributed-shared memory programming template for developing grid applications at an abstract level. Runtime support modules are provided for difficult programming languages and machine platforms. Advanced data services are also introduced to enable users to customize the means in which their application will handle data, and this is especially useful in problems that work on uniquely formatted data, such as data retrieved from specialized databases and in the physical and life sciences. This is the layer that application developers will work with.

The *ALiCE applications and toolkits* layer encompasses the various grid applications and programming models that are developed using ALiCE programming template and it is the only layer visible to ALiCE application users.

2.3. Runtime System

Figure 2 shows ALiCE runtime system. It is driven by the CGS and the DGS components in the ALiCE core layer. The ALiCE CGS component provides the infrastructure for executing grid applications. It adopts a three-tiered architecture, comprising of three main types of entities: *consumer*, *producer* and *resource broker*, as described in the following:

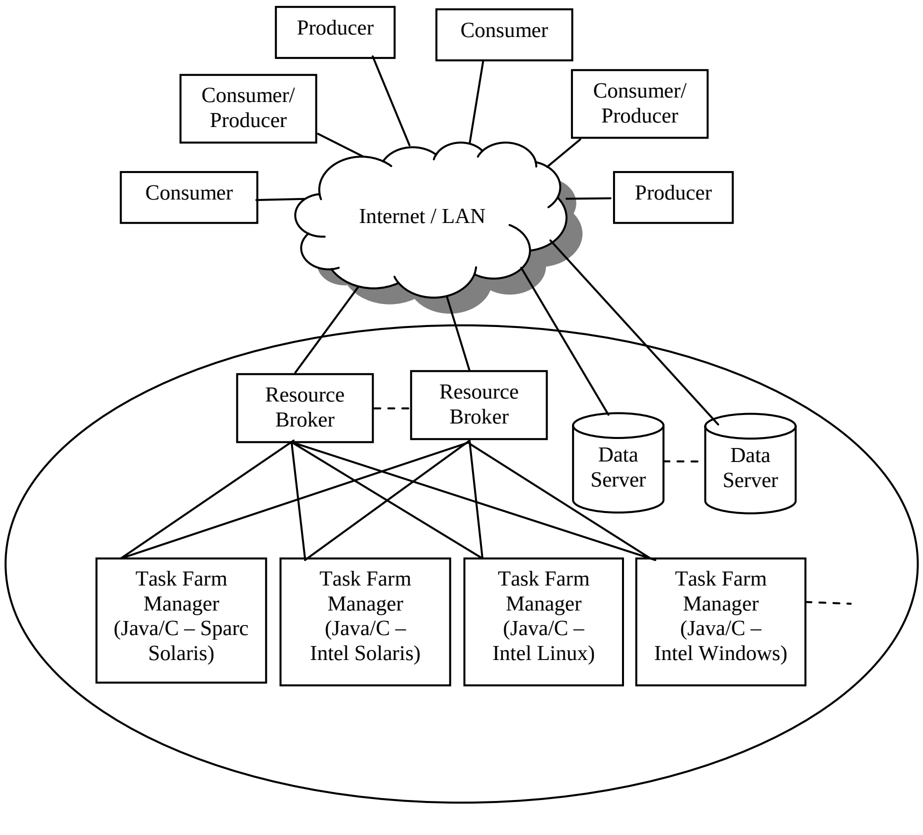
Producer
Consumer
Consumer/ Producer
Consumer/ Producer
Consumer
Producer
Internet / LAN
Resource Broker
Resource Broker
Data Server
Data Server
Task Farm Manager (Java/C – Sparc Solaris)
Task Farm Manager (Java/C – Intel Solaris)
Task Farm Manager (Java/C – Intel Linux)
Task Farm Manager (Java/C – Intel Windows)

Figure 2: ALiCE Runtime System

- ***Consumer***. This submits applications to the ALiCE grid system. It can be any machine within the grid running the ALiCE consumer/producer components. It is responsible for collecting results for the current application run, returned by the tasks executed at the producers, and is also the point from which new protocols and new runtime supports can be added to the grid system.
- ***Resource broker***. This is the core of the grid system and deals with resource and process management. It has a *scheduler* that performs both *application* and *task* scheduling. Application scheduling helps to ensure that each ALiCE application is able to complete execution in a reasonable turnaround time, and is not constrained by the workload in the grid where multiple applications can execute concurrently. Task scheduling coordinates the dissemination of compute tasks, thereby controlling the utilization of the producers. The default task scheduling algorithm adopted in ALiCE is *eager scheduling* [3]. In addition, there are some objective requirements imposed on ALiCE, since one of its goals is to support execution of applications implemented in programming languages other than Java. These applications may be platform and library dependent. The scheduler must therefore select, amongst the producers in the grid, one that runs the most appropriate platform to execute a given application.
- ***Producer***. This is run on a machine that volunteers its cycles to run ALiCE applications. It receives tasks from a resource broker in the form of serialized live objects, dynamically loads the objects and executes the encapsulated tasks. The result of each task is returned to the consumer that submitted the application. A producer and a consumer can be run concurrently on the same machine.
- ***Task Farm Manager***. ALiCE applications are initiated by the Task Farm Manager and the tasks generated are then scheduled by the resource broker and executed by the producers. The task farm manager is separated from the resource broker for two principal reasons. Firstly, ALiCE supports non-Java applications that are usually platform-dependent, and the resource broker may not be situated on a suitable platform to run the task generation codes of these applications. Secondly, for reasons of security and fault tolerant the execution of alien code submitted by consumers is isolated from the resource broker. Each task farm manager runs either Java code or code compiled for the platform that the task farm manager offers.

The ALiCE DGS component comprises of an entity called *data server*, which is run on a machine that stores data files required for the execution of ALiCE applications. A task can obtain a reference to access a data file required for the application. With this reference, the task can write to the file or read chunks of any size from it. The Data Services component allows users to define the methods of handling individually formatted data, thereby customizing the handling of data by the data server. This is to accommodate data files may not be treated uniformly across application domains. For instance, the smallest unit of data in bioinformatics is a protein or genome sequence [29].

In a typical scenario, a user launches an ALiCE application at a consumer, which then submits the application codes to a resource broker in the system. The resource broker directs an application to an appropriate task farm manager that supports the required programming language and platform. The task farm manager initiates the application and creates a pool of tasks. Task references are returned to the resource broker, which schedules the tasks for execution on producers. Results of task execution are then returned to the consumer for visualization. Connections between the system entities are established over a LAN if the environment is a cluster grid or an Intranet, and through the Internet if the grid system is deployed over a WAN.

2.4. Object Communication

The ALiCE communications system supports the migration of codes, data and results of task execution between the runtime entities via space. All communications are via objects and supported by ONTA. When an ALiCE entity (consumer, producer, resource broker) intends to send a live object/code block to another entity, the ONTA service packages it into an archive file, notifies the receiving entity with a reference to the file, downloads the file to the receiving entity and dynamically loads the object from the downloaded archive. ONTA offers the ALiCE core developer an API to serialize and save objects with the associated classes, thus implementing object persistence across the grid. It also facilitates the addition of new protocols into the grid system. Figure 3 illustrates the ONTA mechanism.

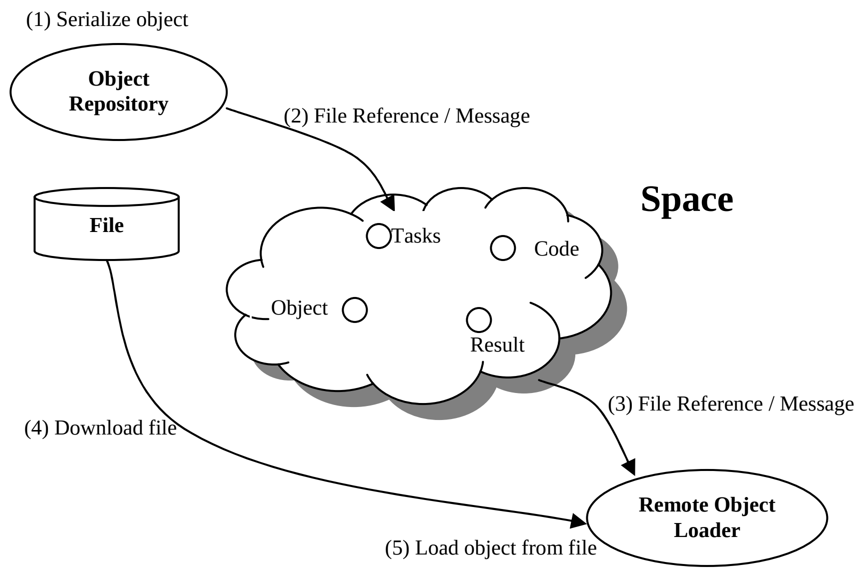

Figure 3: Object Transfer using ONTA

ONTA comprises of five main components:

a. *Object writer*. This is responsible for serializing objects and codes to be transported, creating new .jar archive files and finally placing the serialized objects/codes into these archives.
b. *Object repository*. This stores the .jar archive files created by the Object writer. Whenever new archives are being introduced, it advertises them in the space with references to the archives. Besides objects and codes, it also disseminates newly introduced protocols to the other platforms.
c. *Remote object loader*. This retrieves archive file references from the space and downloads the target files in the Object repository of the sender platform. It also retrieves any messages sent from other machines destined for the platform it is running on.
d. *Object loader*. This restores saved objects from archive files downloaded by the Remote object loader, and sends it to the part of the application running on its platform.
e. *File manager*. This organizes file naming and storage in the Object repository on the local disk. *Protocols* which are moved around the system when needed.

The Object writer, Object repository and File manager are *sender* components, while the other two are *receiver* components.

3. Implementation

ALiCE is scalable and comprises of modular components developed using ubiquitous and easy-to-use object-based Java technologies, including Jini and JavaSpaces [21], thereby providing for *cross-platform portability*, *extensibility* and *scalability*. Each ALiCE producer runs a copy of the JVM, allowing different platforms in the grid to share executables.

Jini defines a runtime infrastructure that unifies all JVMs into a single virtual network, enabling homogeneous hardware devices to plug in to form *decentralized communities*. JavaSpaces, a special service of Jini, is a simple, expressive, and powerful technology with the goal of reducing development time in building distributed applications. All processes are loosely coupled across the network, communicating and synchronizing their activities using a persistent object store called a *space*. All ALiCE entities communicate through JavaSpaces. To the best of our knowledge, ALiCE is the *first* grid-computing project that is developed using Sun's Java-Jini and JavaSpaces.

ALiCE provides an alternative communications platform using GigaSpaces [16]. GigaSpaces Synchronization and Coordination Platform is a software infrastructure for information collaboration platform for Enterprise Distributed Applications and Web Services. The major difference between JavaSpaces and GigaSpaces is that the former provides a logical distributed-shared memory [22] but the latter implements distributed-shared memory by coupling together several spaces hosted at different machines. Our tests show that using GigaSpaces results in better performance and reliability than JavaSpaces.

4. Grid Programming

Grid environments are characteristically distributed and dynamic [23]. ALiCE sets out to provide an effective programming model to facilitate the development of grid applications and higher-level specialized programming models. In the ALiCE paradigm, large computations are decomposed into smaller tasks that are then distributed among producers in the network to exploit parallelism as best as possible to achieve a reasonable amount of speedup.

ALiCE adopts the *TaskGenerator-ResultCollector* programming model. This model comprises of four main components: *TaskGenerator*, *Task*, *Result* and *ResultCollector*. The consumer first submits the application to the grid system in the form of a .jar file encapsulating the application codes. The *TaskGenerator* running at a task farm manager machine generates a pool of *Task*s belonging to the application. These *Task*s are then scheduled for execution by the resource broker and the producers download the tasks from the task pool. The results of the individual executions at the producers are returned to the resource broker as *Result* object. The *ResultCollector*, initiated at the consumer to support visualization and monitoring of data collects all *Result* objects from the resource broker. For batch job, result objects are collected at the resource broker.

Parallel applications development are written using ALiCE programming template. The template allows the programmers to transparently exploit the distributed nature of the ALiCE grid, i.e., without prior knowledge of the underlying technologies for communications, dynamic code linking, etc. The template abstracts methods for generating tasks and retrieving results in ALiCE, leaving the programmers with only the task of filling in the task specifications. Figure 4 shows the ALiCE programming template.

The Java classes comprising the ALiCE programming template are:

a. *TaskGenerator*. This is run on a task farm manager machine and allows tasks to be generated for scheduling by the resource broker. It provides a method process that generates tasks for the application. The programmer merely needs to specify the circumstances under which tasks are to be generated in the main method.
b. *Task*. This is run on a producer machine, and it specifies the parallel execution routine at the producer. The programmer has to fill in only the execute method with the task execution routine.
c. *Result*. This models a result object that is returned from the execution of a task. It is a generic object, and can contain as many user-specified attributes and methods, thus permitting the representation of results in the form of any data structure that are serializable.
d. *ResultCollector*. This is run on a consumer machine, and handles user data input for an application and the visualization of results thereafter. It provides a method collectResult that retrieves a *Result* object from the resource broker. The programmer has to specify the visualization components and control in the collect method.

TaskGenerator Template

```
import alice.consumer.*;
import alice.data.*;
public class TASKGEN_CLASSNAME extends TaskGenerator {
   public TASKGEN_CLASSNAME() {}
   public void init() {
      //Place your initialisation code here
   }

   /* Main method - entry point */
   public void main(String args[]) {
     // This is where the tasks are generated, usually in a loop

     // This should be called for each task
     TASK_CLASSNAME t = new TASK_CLASSNAME();
     process(t);

     // To open a data file, read and write from/to it
     DataFile f = Data.openFile("file_name",this);
     READ_BUFF = f.read(POSITION, LENGTH);
    f.write( WRITE_BUFF, POSITION, LENGTH);

     // To send/receive an object
     OBJECT_CLASSNAME obj = new  OBJECT_CLASSNAME();
     sendObject(obj, "snd_str_id");
     OBJECT_CLASSNAME rcvObj = (OBJECT_CLASSNAME)
                                     requestObject("rcv_str_id");

     // To receive a string message from the result collector:
    String msg = getStringMessage();
  }
}
```

Task Template

```
import alice.consumer.*;
import java.io.*;
public class TASK_CLASSNAME extends Task {
  // Place variables here
  public TASK_CLASSNAME () {
  }

  public Object execute () {
    // This is where you do your computations. The results can be any kind of
    // objects

    // You can generate and send a new task to be produced
    O_TASK_CLASSNAME t = new O_TASK_CLASSNAME();
    process(t);

    // To open a data file, read and write from/to it
    DataFile f = Data.openFile("file_name",this);
    READ_BUFF = f.read(POSITION, LENGTH);
    f.write( WRITE_BUFF, POSITION, LENGTH);

    // To send/receive an object
    OBJECT_CLASSNAME obj = new OBJECT_CLASSNAME();
    sendObject(obj, "snd_str_id");
    OBJECT_CLASSNAME rcvObj =(OBJECT_CLASSNAME)
                                   requestObject("rcv_str_id");
  }
}
```

Result Template

```
import java.io.*;

public class MyResult implements Serializable {
   public DATA_TYPE var;
   public MyResult() {
      var=NULL;
   }
}
```

ResultCollector Template

```
import alice.result.*;
public class RESCOL_CLASSNAME extends ResultCollector {
   // Place Variables Here

   public RESCOL_CLASSNAME() {
   }

   public void collect() {
     // Place here the result collection and processing code to obtain
     // number of results ready call
     int resReady = getResultsNoReady()

      // To get a new result call
     RES_CLASSNAME res = (RES_CLASSNAME)collectResult();
   }
}
```

Figure 4: ALiCE Programming Template

5. Performance

We have developed several distributed applications using ALiCE. These include the distributed Mandelbrot Set Generator, life science applications such as biosequence comparison and progressive Multiple Sequence Alignment [29], satellite image processing [28], N-body problem [19], distributed equation solver, etc. In this paper, we present the results of the DES (*Data Encryption Standard*) key search [32].

DES key search is a mathematical problem, involving the use of a brute force method to identify a selected encryption key in a given key space. A DES key consists of 56 bits that are randomly generated for searching, and 8 bits for error detection. In the algorithm, a randomly selected key, *K*, is used to encrypt a known string into a ciphertext. To identify *K*, every key in the key space is used to encrypt the same known string. If the encrypted string for a certain key matches with the ciphertext, then the algorithm converges and the value of *K* is returned. This problem requires immense computational power as it involves exhaustive search in a potentially huge key space.

The test environment consists of a homogeneous cluster and a heterogeneous cluster with all nodes running RedHat Linux. The 64-node homogeneous cluster (*Cluster I*) consists of dual processors Intel Xeon 1.4GHz processors with 1GB of memory. The nodes are connected by a Myrinet network. The 24-node heterogeneous cluster (*Cluster II*) consists of sixteen nodes Pentium II 400MHz with 256MB of RAM, and eight nodes Pentium III 866MHz with 256MB of RAM. These nodes are connected via 100Mbps Ethernet switch.

Our performance metric is the execution time to search the *entire* key space. As shown in Table 1, the sequential execution time grows exponentially with increasing key sizes. For larger key sizes, the execution time is estimated from the measured execution time obtained from smaller key sizes. The results show that the estimated times match the measured times for the smaller key sizes.

Key Size (bits)	**Execution Time**			
	Xeon 1.4 GHz		**Pentium III 866 MHz**	
	Measured	**Estimated**	**Measured**	**Estimated**
24	1 min	1 min	1 min	1 min
28	18 min	18 min	29 min	29 min
32	4 hr 52 min	4 hr 57 min	7 hr 55 min	7 hr 57 min
36	-	3.5 days	-	6 days
40	-	3 yrs	-	5 yrs
56	-	201,396 yrs	-	328,690 yrs

Table 1: Sequential DES Key Search Performance

The DES key problem can be partitioned into varying number of tasks with a task size measured by the number of keys and its execution time can be estimated using the time from the sequential run. Table 2 shows the task characteristics for varying task sizes and problem sizes. The table was used to select an appropriate task size for the experiments to be carried out in the two grid configurations.

task size (keys)	**32-bit Key**			**36-bit Key**		**40-bit Key**	
	no. of tasks	**Est. Time/Task (secs)**		**no. of tasks**	**est. time/task (secs)**	**no. of tasks**	**est. time/task (secs)**
		cluster I	**cluster II**		**cluster I**		**cluster I**
5,000,000	859	20.8	32.9	13,744	20.8	219,902	430.3
10,000,000	429	41.1	65.4	6,872	43.4	109,951	862.4
30,000,000	143	122.9	196.6	2,291	127.8	36,650	2587.7
50,000,000	86	201.9	322.0	1,374	211.4	21,990	4314.3
100,000,000	43	395.4	641.2	687	420.1	10,995	8629.5

Table 2: Estimated Task Execution Times for Varying Task Sizes

For our experiments conducted, we selected a task size of 50 million keys per task and a problem size of 36-bit keys for Cluster I and 32-bit keys for Cluster II. Table 3 shows the results for 4 to 32 producer nodes. The execution time for key search reduces significantly with increasing number of nodes, resulting in greater speedup.

We define *speedup* as T_s/T_p, where T_s is the execution time of the sequential program and T_p is the execution time of the derived parallel program on *p* processors. As shown in Figure 5, a speedup of approximately 32 is attained for key size 36-bits on Cluster I and 8 for 32-bits on Cluster II. We consider these results highly encouraging, although the performance of key search needs to be further evaluated with more key space sizes and nodes. The effects of using other scheduling algorithms in the resource broker must also be studied, as it may result in different overheads to the execution time.

No. of Producers	**Cluster I (36-bit Key)**	**Cluster II (32-bit Key)**
1 (Est. Sequential)	78 hr 23 min	8 hr 43 min
4	23 hr 36 min	3 hr 43 min
8	11 hr 6 min	2 hr 7 min
10	8 hr 34 min	1 hr 42 min
12	7 hr 21 min	1 hr 26 min
16	5 hr 11 min	1 hr 7 min
32	2 hr 29 min	-

Table 3: Execution Time for Varying Number of Producer Nodes

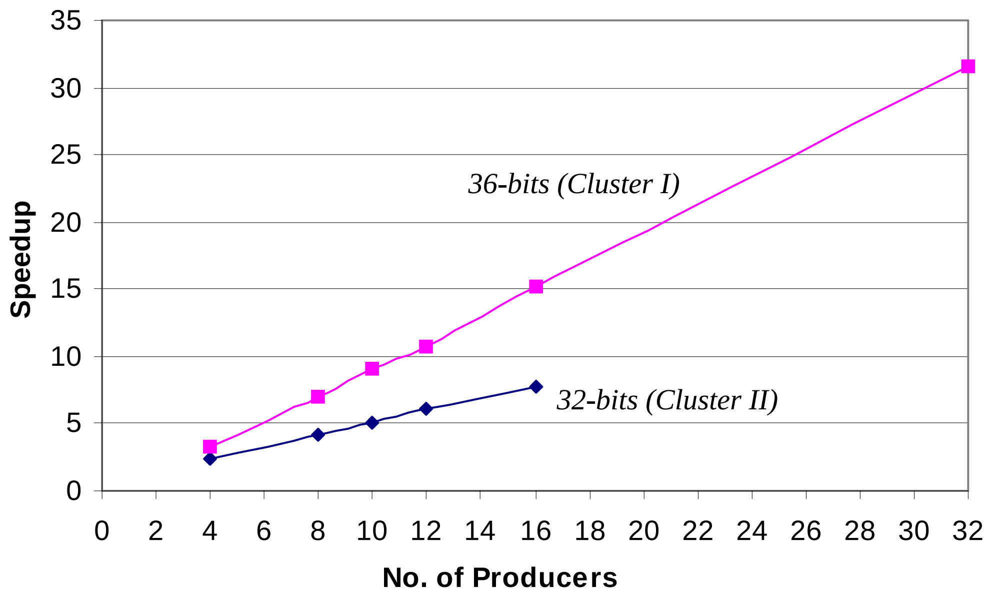

Figure 5: Speedup vs Varying Number of Producers

6. Conclusion and Further Works

We discussed the design and implementation of the Java-based ALiCE grid system. The runtime system comprises of consumers, producers and resource broker. The runtime system supports grid applications written in different programming languages and machine platforms. Parallel grid applications are written using programming template that supports the distributed-shared memory programming model. We presented the performance of ALiCE using the DES key search problem. The result shows that a homogeneous cluster yields greater speedup than on a heterogeneous cluster for the same task size. A homogeneous cluster generally has a better load balance than a heterogeneous cluster which is made up of different platforms and capabilities.

Much work still needs to be done to transform ALiCE into a comprehensive grid computing infrastructure. We are in the process of integrating new resource scheduling techniques and load-balancing mechanisms into the ALiCE core layer to reduce the overhead in running applications [30]. Task migration, pre-emption and check-pointing mechanisms are being incorporated to improve the reliability and fault-tolerance ability of the system. Other ongoing works include implementing QoS techniques to prioritize critical applications.

References

1. Allcock, B., Bester, J., Bresnahan, J., Chervenak, A., Foster, I., Kesselman, C., Meder, S., Nefedova, V., Quesnel, D. and Tuecke, S., Secure Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing, *Proceedings of the IEEE Mass Storage Conference*, April 2001.
2. Baker, M., Buyya, R. and Laforenza, D., Grids and Grid Technologies for Wide-Area Distributed Computing, *International Journal of Software: Practice and Experience (SPE)*, 32(15), Wiley Press, USA, November 2002.
3. Baratloo. A, Karaul. M, Kedem. Z and Wyckoff. P, Charlotte: Metacomputing on the Web, *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, 1996.
4. Buyya, R. and Murshed, M., GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing, *The Journal of Concurrency and Computation: Practice and Experience (CCPE)*, pp. 1-32, Wiley Press, May 2002.
5. Casanova, H. and Dongarra, J., NetSolve: A Network Server for Solving Computational Science Problems, *International Journal of Supercomputing Applications and High Performance Computing*, 11(3), 1997.
6. Czajkowski, K., Foster, I., Karonis, N., Kesselman, C., Martin, S., Smith, W. and Tuecke, S., A Resource Management Architecture for Metacomputing Systems, *Proceedings of the IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 62-82, 1998.
7. De Roure, D., Baker, M. A., Jennings, N. R. and Shadbolt, N. R., The Evolution of the Grid, Research Agenda, UK National eScience Center, 2002.
8. Fitzgerald, S., Foster, I., Kesselman, C., von Laszewski, G., Smith, W. and Tuecke, S., A Directory Service for Configuring High-Performance Distributed Computations, *Proceedings of the 6th IEEE Symposium on High-Performance Distributed Computing*, pp. 365-375, 1997.
9. Foster, I., Computational Grids, Morgan Kaufmann Publishers, 1998.
10. Foster. I and Kesselman. C, Globus: A Metacomputing Infrastructure Toolkit, *International Journal of Supercomputing Applications*, 11(2), pp 115-128, 1997.
11. Foster, I., Kesselman, C., Lee, C., Lindell, R., Nahrstedt, K. and Roy, A., A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation, *Proceedings of the International Workshop on Quality of Service*, 1999.
12. Foster, I., Kesselman, C., Nick, J. M. and Tuecke, S., The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, *Proceedings of CGF4*, February 2002, http://www.globus.org/research/papers/ogsa.pdf.
13. Foster, I., Kesselman, C. and Tuecke, S., The Anatomy of the Grid: Enabling Scalable Virtual Organizations, *International Journal of Supercomputer Applications*, 15(3), 2001.
14. Foster, I., Kesselman, C. and Tuecke, S., The Nexus Approach to Integrating Multithreading and Communication, *Journal of Parallel and Distributed Computing*, 37, pp. 70-82, 1996.
15. Frey, J., Tannenbaum, T., Foster, I., Livny, M. and Tuecke, S., Condor-G: A Computation Management Agent for Multi-Institutional Grids, *Journal of Cluster Computing*, 5, pp. 237-246, 2002.
16. GigaSpaces Platform White Paper, GigaSpaces Technologies, Ltd., February 2002.
17. Grid Computing Info Center (GRID Infoware), http://www.gridcomputing.com.
18. Grimshaw, A. and Wulf, W., The Legion Vision of a Worldwide Virtual Computer, *Communications of the ACM*, 40(1), January 1997.
19. Ho, D. P., Y M Teo, J P Gozali, Solving the N-body Problem on the ALiCE Grid System, 7th Asian Computing Science Conference, Lecture Notes in Computer Science 2250, pp. 87-97, Springer-Verlag, Hanoi, Vietnam, December 2002.
20. Hoschek, W., Jaen-Martinez, J., Samar, A., Stockinger, H. and Stockinger, K., Data Management in an International Data Grid Project, *Proceedings of the 1st IEEE/ACM International Workshop on Grid Computing (GRID2000)*, Bangalore, India, pp. 17-20, December 2000.
21. Hupfer, S., The Nuts and Bolts of Compiling and Running JavaSpaces Programs, Java Developer Connection, Sun Microsystems, Inc., 2000.
22. Itzkovitz, A. and Schuster, A., Distributed Shared Memory: Bridging the Granularity Gap, *Proceedings of the First ACM Workshop on Software Distributed Shared Memory (WSDSM)*, Greece, June 1999.
23. Lee, Matsuoka, Talia, Sossman, Karonis, Allen and Thomas, A Grid Programming Primer Programming Models Working Group, Grid Forum 1, Amsterdam, 2001.
24. Lewis, M. and Grimshaw, A., The Core Legion Object Model, *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, August 1996.
25. Project JXTA White Paper, Sun Microsystems, Inc., http://wwws.sun.com/software/jxta/jxta-doc.html.
26. RSA130 Collaboration – FAFNER: Factoring via network-Enabled Recursion, http://cooperate.com/cgi-bin/FAFNER/factor.pl.
27. SETI@Home: Search for Extraterrestrial Intelligence at Home, http://setiathome.ssl.berkeley.edu.

28. Teo., Y.M., S.C. Tay and J.P. Gozalijo, Geo-rectification of Satellite Images using Grid Computing, Proceedings of the International Parallel & Distributed Processing Symposium, IEEE Computer Society Press, Nice, France, April 2003.
29. Teo Y.M. and Ng Y.K., *Progressive Multiple Biosequence Alignments on the ALiCE Grid*, Proceeding of the 6th International Conference on High Performance Computing for Computational Science, Springer Lecture Notes in Computer Science Series, xx, Spain, June 28-30, 2004 (accepted for publication).
30. Teo Y.M., X. Wang, J.P. Gozali, A Compensation-based Scheduling Scheme for Grid Computing, Proceedings of the 7th International Conference on High Performance Computing, IEEE Computer Society Press, Tokyo, Japan, July 2004 (submitted).
31. Wahbe, R., S. Lucco, T.E. Anderson, and S. L. Graham, Efficient Software-based Fault Isolation, *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, Asheville, North Carolina, USA, pp. 202-216, December 1993.
32. Wiener, M., Efficient DES Key Search, Practical Cryptography for Data Internetworks, William Stallings, IEEE Computer Society Press, pp. 31-79, 1996.